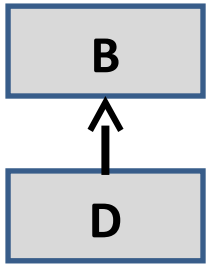
Wprowadzenie w dziedziczenie

Klasa D dziedziczy klasę B:



klasa bazowa

klasa pochodna

- **Klasa B – klasa bazowa (base class), klasa D – klasa pochodna (derived class).**

- **Najpierw jest tworzona klasa bazowa, która zawiera elementy wspólne dla wszystkich obiektów klas pochodnych. Klasa bazowa to najbardziej ogólny opis.**

- **Klasy, utworzone na podstawie klasy bazowej, nazywamy klasami pochodnymi. Klasa pochodna ma wszystkie cechy klasy bazowej oraz dodatkowe rozszerzenia.**

Przykład W4_0:

```
//deklaracje klasy bazowej
class B
{
//protected:
        int i;
 public:
        void set_i(int n);
        int get_i();
};

//deklaracje klasy pochodnej
class D : public B
{
        int j;
 public:
        void set_j(int n);
        int mul();
};
```

Deklaracje klasy pochodnej:

class D *: public* B

class nazwa_klasy_pochodnej : *public* nazwa_clasy_bazowej

Kluczowe słowo *public* oznacza:

- **składowe publiczne klasy bazowej stają składowymi publicznymi klasy pochodnej**

- **składowe prywatne klasy bazowej pozostają niedostępnymi dla klasy pochodnej – klasa pochodna nie ma dostępu bezpośredniego do tych składowych – dane prywatne klasy bazowej mogą być udostępnione w klasie pochodnej tylko przez metodę publiczne klasy bazowej (albo poprzez deklaracje *protected*).**

W przykładzie podanym:

- **Funkcja get_i() jest metodą klasy bazowej, przecież jest bezpośrednio dostępną w zasięgu klasy pochodnej (metoda D::mul()) – składowe publiczne klasy bazowej stają składowymi publicznymi klasy pochodnej.**

- **Bezpośredni dostęp do zmiennej prywatnej klasy bazowej *B::i* w zasięgu klasy pochodnej (wewnątrz klasy pochodnej) jest wzbroniony – zasada hermetyzacji (inkapsulacji). Więc dla udostępnienia tej zmiennej używamy metodą get_i() (albo zmieniamy specyfikator dostępu z *private* na *protected*).**

Obiekt klasy pochodnej składa się ze składowych klasy pochodnej oraz z składowych klasy bazowej (przykład W4_0 – debugger).

Ogólna postać dziedziczenia:

```
class nazwa_clasy_pochodnej : specyfikator_dostępu nazwa_clasy_bazowej
{
        //ciało nowej klasy
};
```

specyfikator_dostępu – public [private] [protected]

Będziemy na razie rozważali tylko specyfikator dostępu *public* .

Wskaźniki do obiektów

Definicje wskaźników do obiektów:

```
MY_CLASS ob;          //deklaracja obiektu – powoduje wydzielenie pamięci
                      // dla całego obiektu, wywołuje konstruktor
MY_CLASS *ptr_ob;     // deklaracja wskaźnika do obiektu. Wydziela 4 B
                       //pamięci dla wskaźnika. Wskaźnik jest na razie
                       //nie zainicjowany
ptr_ob = &ob;         //Teraz ptr_ob wskazuje na obiekt ob.

ob.i = ……             //Dostęp do publicznej składowej obiektu ob
ptr_ob->i = …          //To samo, dostęp przez wskaźnik

ob.show();            //wywołanie metody klasy
ptr_ob->show();       //to samo, przez wskaźnik

MY_CLASS obb[20];     //deklaracja tablicy obiektów
ptr_ob = obb;          //ptr_ob wskazuje na zerowy element  tablicy
ptr_ob++;              //ptr_ob wskazuje na pierwszy element tablicy
ptr_ob = &obb[0];     //ptr_ob wskazuje na zerowy element  tablicy
ptr_ob += 10;          //ptr_ob wskazuje na 10 element tablicy
```

```
class area_cl
{
        double width;
public:
        double get_width() { return width; }
        void set_width(double a) { width = a; }
};

int  main()
{
        area_cl obb[20];
        int it;
        for(it=0; it<20; it++)
        {
                        obb[it].set_width((double)(it));
        }
        area_cl *ptr_ob = NULL;
        ptr_ob = obb;

        for(it=0; it<20; it++, ptr_ob++)
        {
                        if(fabs(ptr_ob->get_width()-obb[it].get_width()) > 1.0e-9)
                                        cout << "error\t";
        }

        return 0;
}
```

Klasy, struktury, unie

- **W C++ niema różnicy podstawowej pomiędzy klasą a strukturą. Struktura może zawierać nie tylko dane, a i funkcje. Różni się struktura od klasy tym, że domyślne składowe klas są prywatne, a struktur – publiczne. Żeby element struktury zadeklarować jako składową prywatną, należy użyć kluczowe słowo *private*. Chocza struktura może zawierać i metody, zwykłe programiści używają struktury dla przedstawienia danych, a dla przedstawienia obiektów – klasy.**

- **Instrukcja union podobnie jak struktura może być wykorzystywana do definiowania klas – unie mogą zawierać zarówno funkcje składowe, jak i dane. Mogą mieć konstruktor i destruktor. Najważniejsze jest umieszczanie wszystkich danych w tym samym obszarze pamięci. Składowe unii domyślne są publiczne.**

- **Na różnice od klas unie nie mogą dziedziczyć ani żadnych klas, ani typów. Unia nie może być klasą bazową.**

- **Składowymi unii nie mogą być:**
  - **funkcje wirtualne**
  - **zmienne statyczne**
  - **zmienne referencyjne**

- obiekt z przeciążonym operatorem =
- obiekty, dla których w sposób jawny zdefiniowano konstruktor lub destruktor

➢ **Unie anonimowe – nie zawierają nazwę typu. Nie można deklarować obiektów typu, definiowanego przez taką unię. Unie anonimowe informują kompilator, że ich zmienne składowe będą znajdowali się w tym samym obszarze pamięci. Dostęp do składowych unii anonimowej odbywa się bezpośrednie:**

```
union
{
        int i;
        char ch[4];
}

i = 4;
ch[0] = 'X';
```

- **Globalne unie anonimowe mogą być zadeklarowane tylko jako static**

- **Unii anonimowe nie mogą zawierać składowych prywatnych**

- **Nazwy składowych unii anonimowej nie powinni konfliktować z innymi nazwami zmiennych w zasięgu deklaracji.**

Przykład Union_1

Funkcje wplatane (inline)

- **To są krótkie funkcje, które w rzeczywistości nie są wywoływane. Zamiast tego ich ciało pozostaje wbudowano w kod programu w miejscu jej wywołania.**

- **Taka technika nadaje możliwość obejść mechanizm wywołania funkcji, związany z procedurą tworzenia steku argumentów formalnych, przekazania wartości argumentów aktualnych, zwracania wartości (return value), itp. Dla tego inline – funkcji będą wykonane znacznie szybszej od wywołania zwykłego. To jest zaletą funkcji wplatanych.**

- **Wadą funkcji wplatanych jest to, że jeśli kod takich funkcji jest długi i funkcje wplatane są wywołane zbyt często, objętość programu będzie drastycznie rosła.**

Definicje inline funkcji:

```
#include <iostream>
using namespace std;

inline int even(int x)
{
        return (x%2);
}

int main()
{
        if (even (10)) cout << ”10  jest wartość nieparzysta” << endl;  //(10%2) = 0 – cout nie wykon.
        if (even (11)) cout << ”11  jest wartość nieparzysta” << endl;  //(11%2) = 1 – cout wykon.
}
```

- **Inline funkcja musi być zdefiniowaną przed pierwszym wywołaniem. (Inaczej kompilator nie wie, który kod ma wplatać).**

- **Specyfikator inline nie jest poleceniem, tylko żądaniem – kompilator może jego ignorować. Przy ignorowaniu inline funkcja będzie wywołana jako zwykła.**

- **W zależności od typu kompilatora mogą być szereg ograniczeń:**
  - **funkcja nie musi być rekursywną**
  - **nie musi zawierać zmienne statyczne**
  - **nie musi mieć instrukcje pętli lub switch lub goto**

- **Jeśli takie ograniczenie istnieje, zamiast inline będzie generowane wywołanie zwykłe.**

- **Funkcje – składowe klasy również mogą być wplatane**

```
#include <iostream>
using namespace std;

class samp
{
	int i;
public:
	samp(int a);
	int get_i();
};

samp::samp(int a)
{
	i = a;
}

inline int samp::get_i()
{
	return i;
}
```

- **Funkcji inline można używać w deklaracje klasy:**

```
class  samp
{
      int  i, j;
public:
      samp(int  a, int b)        //kilka linij kodu inline - funkcje
      {
           i = a;
           j = b;
      }

      int get_i() { return i; }     //inline funkcja umieszczona w jedną linie kodu
      int get_j() { return j; }     //to jest najbardziej wspierany styl śród programistów
}
```

Przy takim stylu specyfikator inline jest pominięty.

- **Funkcje inline można przeciążyć, przy tym wszystkie realizacje mają być też inline**

Przypisanie obiektów

- **Jeśli typ dwóch obiektów jest taki samy, to jeden obiekt można przypisać drugiemu. Domyślnie przypisanie jednego obiektu drugiemu oznacza, że dane obiektu, znajdującego się z prawej strony, zostaną skopiowane do obiektu znajdującego się z lewej strony bit po bicie.**

```
class myclass
{
        int i;
public:
        void set_i(int ii);
        int get_i();
};

int main()
{
        myclass ob1, ob2;
        ob1.set_i(10);
        ob2 = ob1;
        cout << ob1.get_i() << ”\t” << ob2.get_i() << endl;
……..
}
```

WYDRUK:

10 10

Przypisanie obiektów

- **Jeśli dane klasy zawierają wskaźnik(i) do obiektów dowolnego typu (do tablic typów wbudowanych), przypisanie domyślne powoduje błąd –**

przykład W4_1

- **Dla tego, żeby uniknąć takiej sytuacji, operator przypisania trzeba przeciążyć. Będziemy rozważali to później.**

Przekazywanie obiektów do funkcji

- **Obiekty mogą być przekazywane do funkcji w ten sam sposób, co zmienne innych typów. Domyślnie, obiekty są przekazane do funkcji przez wartość. Oznacza to, że tworzona jest kopia obiektu, następnie przekazywana do funkcji. Więc dowolne zmiany danych, dokonywane funkcją, to są zmiany kopii obiektu, a nie samego obiektu, i nie wpływają na sam obiekt.**

- **Funkcji można przekazać adres obiektu. W takim razie jest tworzona kopia wskaźnika do obiektu, która zawiera adres obiektu. W funkcji dostęp do**

składowych obiektu będzie dokonywany przez adres obiektu, więc wszystkie zmiany pozostaną wprowadzone na danych samego obiektu.

Przykład W2_0:

- **Przekazanie obiektu funkcji przez wartość powoduje tworzenie kopii tego obiektu – powstaje nowy obiekt. Przy wyjściu z ciała funkcji kopia obiektu wychodzi poza zasiąg deklaracji – kopia powinna być zniszczona.**

! Przy tworzeni kopii obiektu konstruktor dla kopii się nie wywołuje.
Przy wyjściu z ciała funkcji wywołuje się destruktor kopii.

- **Przy tworzeni kopii obiektu wywołanie konstruktora dla kopii powodowało by inicjowanie danych zamiast przekazania ich wartości od oryginała obiektu do kopii. Dla tego konstruktor kopii się nie wywołuje.**

- **Kopia obiektu jest tworzona w steku, więc kiedy działanie funkcji się skończy (to się zwykłe odbywa kiedy funkcja zwraca swoją wartość), kopia obiektu podlega zniszczeniu – dla tego będzie wywołany destruktor.**

- **Przykład W2_1:**

Domyślnie (jeśli nie pozostał zdefiniowany przez programista tak zwany konstruktor kopiujący) kopia jest tworzona na poziomie bitowym – jest dokładnym duplikatem oryginału.
Ten fakt może powodować istotne problemy, związane z efektami ubocznymi.

- **Jeśli na przykład obiekt, przekazywany przez listę argumentów funkcji, dynamicznie alokuje pamięć, a następnie, gdy jest zniszczony, zwalnia ją, to przekazana do funkcji kopia będzie podczas wywołania destruktora zwalniała tą samą pamięć. W rezultacie oryginalny obiekt pozostanie uszkodzony.**

- **Przykład W2:**

- **Istnieje 2 wyjścia z tej sytuacji.**

- **Pierwsze – to jest przekazanie wskaźnika do obiektu przez listę argumentów funkcji. Wtedy nie wynika omówionych powyżej efektów ubocznych. Oprócz tego, to jest dość szybki i skuteczny sposób wywołania funkcji: przy wypełnieni steku jest tworzona kopia tylko wskaźnika – zmiennej 4 B, a nie całego obiektu, który może zawierać duży obszar danych. Wadą takiego podejścia jest to, że dostęp przez wskaźnik nie chroni obiekt od zmian,**

dokonywanych w funkcji. Kontrole takiego kodu wymaga wysokiej uwagi od programisty, dla tego że zmiany obiektu mogą być dokonywane na dolnym poziomie.

- **Drugie wyjście – to tworzenie tak zwanych konstruktorów kopiujących. Będziemy to rozważali później.**

- **Istnieją przypadki, kiedy do funkcji trzeba przekazać tylko wskaźnik do obiektu – przykład „allocator”.**